A Testability Transformation Approach for State-Based Programs

AbdulSalam Kalaji, Robert M Hierons and Stephen Swift
School of Information Systems, Math and Computing
Brunel University, Uxbridge, UB8 3PH, UK
{abdulsalam.kalaji, rob.hierons, stephen.swift}@brunel.ac.uk

Abstract

Search based testing approaches are efficient in test data generation; however they are likely to perform poorly when applied to programs with state variables. The problem arises when the target function includes guards that reference some of the program state variables whose values depend on previous function calls. Thus, merely considering the target function to derive test data is not sufficient. This paper introduces a testability transformation approach based on the analysis of control and data flow dependencies to bypass the state variable problem. It achieves this by eliminating state variables from guards and/ or determining which functions to call in order to satisfy guards with state variables. A number of experiments demonstrate the value of the proposed approach.

1. Introduction

Errors in software can lead to undesired outcomes and testing is therefore a crucial stage. However, manual testing is expensive, error-prone and time consuming hence automation is very desirable.

SBT approaches such as evolutionary testing (ET) [1] have received attention due to their efficiency in deriving test data automatically, however, their applications were largely focused on structured programs where the input domain of a test target is explored to select a set of input values according to a given test criterion e.g., statement coverage. The exploration is steered by evaluation information represented by a *fitness function*. For example, Wegener et al. [2] described a fitness function (Equation 1) in the presence of nested IF statements that comprises two components: a *branch distance* [3] and the *approach level* (Equation 2) to measure how close a particular input was to executing the target branch that is missed and how many critical nodes are away from the target respectively. The critical node is a branching node at which the path control flow may divert (see Fig. 1). Since it is necessary to contrast how many conditions were achieved by a specific input, the branch distance of each IF statement is normalized to a value in the range of [0..1] (Equation 3).

$$fitness = approach\ level + norm\ (branch\_distance) \quad (1)$$
$$approach\_level = 1 - NumCriticalNodeFromTarget \quad (2)$$
$$norm\ (branch\_distance) = 1 - 1.05^{-branch\_distance} \quad (3)$$

The existence of state variables in the presence of function calls can cause problems when using SBT approaches. The main effect is that the fitness function is unable to direct the search towards the desired input values. Thus, the performance of an SBT approach is likely to degenerate to that of random search.

In the literature, some techniques, cited in [4], studied the problem of test data generation from subjects with state behavior. However, an efficient and easy test data generation approach remains a requirement. Thus, the aim of this paper is to benefit from the efficiency and flexibility introduced by testability transformation (TeTra) approaches [5] and reformulate the state variable problem as a TeTra problem. Applying TeTra to a program with state behavior was recently highlighted by Harman [4] as an open research problem.

The approach presented in this paper aims to address the problem described as: Given a target function with state variable problems, transform the test target so that an ET approach can automatically generate a set of test data that exercises this function

The primary contributions of this paper are the following: (1) It proposes a TeTra approach to bypass state variables problems. (2) It provides a method to suggest when a TeTra is likely to be required. (3) The approach can be generally applied to similar problems in a program with functions and global variables.

2. The Proposed Approach

The approach of Wegener et al. [2] described in the previous Section is efficient when a given target function is independent e.g., it is not control or data dependent on other previous functions. Nevertheless,

978-0-7695-3675-0/09 $25.00 © 2009 IEEE
DOI 10.1109/SSBSE.2009.14

```
1- if (x > y)
2-  if (x == 0)
3-    // Target
```

Nodes 1 and 2 are critical nodes

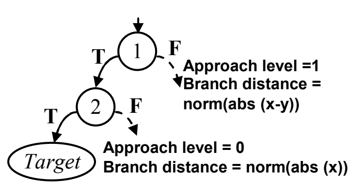

1
T
F
Approach level =1
Branch distance = norm(abs (x-y))
2
T
F
Approach level = 0
Branch distance = norm(abs (x))
Target

Figure.1 An example of a fitness calculation

when such a dependency exists, it is not always possible to consider only the target function.

In order to apply TeTra, we classify an assignment statement *op* to a variable *v* in a function *f* to four types: { op^{vp}, op^{vv}, op^{vc}, $op^{v\pm c}$} which denote that *v* is assigned a value that depends on a parameter, another variable, a constant and itself and a constant respectively. Also, we classify a guard *g* with a guard operator *gop* ∈ {<, >, ≠, =, ≤, ≥ } in an *f* to five types: {g^{pc}, g^{pp}, g^{pv}, g^{vv}, g^{vc}} which denote a comparison among: parameters and constant, parameters only, parameters and variables, variables only and variables and constant respectively. Based on the above classifications, we can distinguish two types of functions: affecting and affected-by functions.

Definition 1: In a given program with *n* functions, f_i is an affecting function within this program if f_i has an *op* ∈ {op^{vp}, op^{vc}, op^{vv}, $op^{v\pm c}$} to *v* and there exists a guarded function f_j, where $0 \leq i < j \leq n$, f_j has a guard *g* ∈ {g^{vv}, g^{vc}} over *v* and the statements at *op* in f_i and *g* in f_j form a definition-use (*du*) pair for *v*.

Definition 2: An f_j is an affected-by function within a program if f_j has *g* ∈ {g^{vv}, g^{vc}} over *v* and there exists an affecting function f_i, where $0 \leq i < j \leq n$, over *v* and the statements at *op* in f_i and *g* in f_j form a *du* pair for *v*.

From Definition 2, an affected-by f_j is always data dependent on the corresponding affecting f_i. f_j is also control dependant on f_i if f_i has a guard that controls its assignment operations that affect f_j. Based on Definitions 1 and 2, we can distinguish four cases in which a target function requires a transformation.

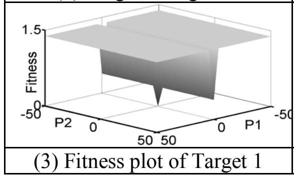
Case 1: the problem occurs between a pair of affected-

```
//Case study 1
int x,y;
void reset()
{x = 0; y = 0;}
void t1 (int P1,P2)
{if (P1==0) x = 10;
 if (P2==0) y = 10;}
void target ()
{if (x>=10 && y>=10)
    //Target 1 }
```

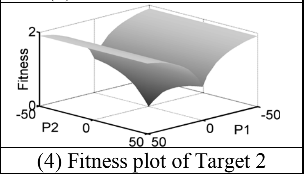
(1) Program fragment

```
//Case study 1- TeTra
int x,y;
void reset()
{ x = 0; y = 0;}
void target (int P1,P2)
{if (P1==0){x =10;
  if (P2==0){y =10;
   if (x>=10 && y>=10)
    //Target 2
}}}
```

(2) Transformed version



(3) Fitness plot of Target 1



(4) Fitness plot of Target 2

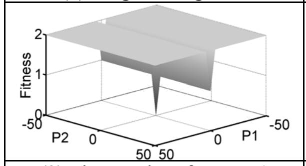
Figure.2 TeTra applied to the first case

```
//Case study -2-
int x,y;
void reset()
{ x = 0; y = 0; }
void t1 (int P1)
{if (P1 != 0) x= 100;
 else x= P1;}
void t2 (int P2)
{if (P2 != 0) y=100;
 else y = P2;}
void target()
{if (x ==0 && y ==0)
 //Target 1 }
```

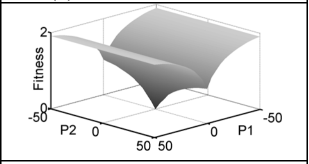
(1) Program fragment

```
//Case study 2 -TeTra
int x,y;

void reset()
{ x = 0; y = 0;}

void target(int P1,P2)
{
if (P1 ==0 && P2 ==0)
    //Target 2
}
```

(2) Transformed version



(3) Fitness plot of Target 1



(4) Fitness plot of Target 2

Figure.3 TeTra applied to the second case

by and affecting functions (f_j, f_i) where *op* in f_i ∈ {op^{vv}, op^{vc}} and f_j is control dependent on f_i.

Fig. 2 shows a case study that describes how TeTra is applied. The target function (*target*) is control dependent on task *t1* which has guards that control its assignments (op^{vc}). A sequence of calls to *reset*→*t1*→*target* does not necessarily achieve the *target* guards (g^{vc}). The fitness landscape of the original program is plotted in Fig. 2-3. Due to the flat region of this landscape, the search does not receive adequate information and relies only on chance to hit the target. Fig. 2-2 shows the transformed version of the *target* task. Since the assignments of *t1* are controlled by its input parameters, these are required on the *target* task. Also, the true branches of *t1* predicates are considered since they lead to assigning variables *x* and *y* the required values. Now, the fitness landscape of Target 2 (see Fig. 2-4) has a clear downward surface and provides adequate guidance.

Case 2: The problem occurs in a pair of affected-by and affecting functions (f_j, f_i) where *op* in f_i ∈ {op^{vp}} and f_j is control dependent on f_i.

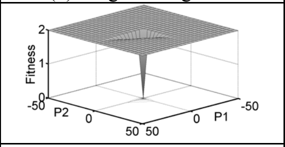
Compared to Case 1, this case has the input parameters of the affecting function referenced by the state variables that appear in the target function guards. Fig. 3 shows a case study in which a *target* task is affected by two functions *t1* and *t2* and these functions have guards that control their assignments. Furthermore, the input parameters of *t1* and *t2* are referenced by the state variables in the *target* task guards. A sequence of calls to *reset*→*t1*→*t2*→*target* does not always lead to the *target* task being exercised. Fig. 3-3 shows that the fitness landscape of the original program is flat. The transformed version of the *target* task is shown in Fig-3-2. The input parameters and the assignment enabling predicates of *t1* and *t2* are

```
//Case study 3
int x,y;
void reset()
{x = 0; y = 0;}
void t1 (int P1,P2)
{if (P1 >= 0){
  x=x+P1*2; y=y+P1;}
 else{x=100; y=100;}
 if (P2 >= 0){
  x=x+P2*3; y=y+x;}
 else{x=100; y=100;}}
void target()
{if (x==0 && y==0)
  //Target 1 }
```

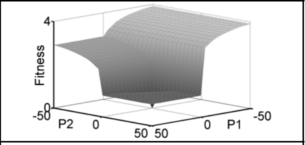
(1) Program fragment

```
//Case study 3 - TeTra
int x,y;

void reset()
{ x = 0; y = 0;}

void target (int P1,P2)
{
if (P1 >= 0){
 if (P2 >= 0){
  if (P1*2 + P2*3==0){
   if (P1*3 + P2*3==0){
        //Target 2
}}}}}
```

(2) Transformed program



(3) Fitness plot of Target 1



(3) Fitness plot of Target 2

Figure.4 TeTra applied to the third case

embedded in the transformed version. Also, the state variables of the original *target* task are replaced by the input parameters that they reference. As observed in Fig. 3-4, the fitness landscape of Target 2 provides the search with enough guidance.

Case 3: The problem exists between a pair of affected-by and affecting functions (f_j, f_i) where *op* in $f_i \in \{op^{vv}, op^{vp}, op^{vc}\}$ and f_j is control dependent on f_i.

This case can be seen as a generalisation of Case 1 and Case 2. However, the main difference is that the affecting function assignments are complicated by many types of assignments. Consequently, bypassing the state variables in this case is not a straight forward process. This problem can be transformed by applying amorphous slicing [6] for the state variables of the target function.

Fig. 4-1 presents a case study in which the *target* task has two state variables which are assigned values in *t1*. A sequence of calls to *reset*$\rightarrow$*t1*$\rightarrow$*target* is unlikely to solve the problem. The original fitness landscape plotted in Fig. 4-3 is almost flat and provides insufficient guidance. In Fig. 4-2, the transformation is applied to replace the state variables *x* and *y* by expressions that reference parameters. The fitness landscape of the transformed version provides the search with adequate guidance as shown in Fig. 4-4.

Case 4: The problem occurs between a pair of affected-by and affecting functions (f_j, f_i) where *op* in $f_i \in \{op^{v\pm c}\}$ and f_j is control dependent on f_i.

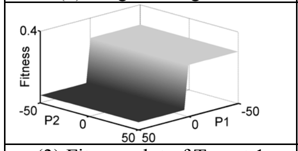
This problem is likely to exist when a state variable in the target function has the role of a counter. For such a case, it is necessary to determine which and how many calls to be made to other functions before calling the target one. Fig. 5-1 shows a case study of two functions where the *target* task is control dependent on the affecting *t1*. As shown in Fig. 5-3, the fitness

```
//Case study 4
int x,y;
void reset()
  { x = 0; y = 0;}
void t1 (int P1,P2)
{if (P1 >= 0)
   x=x+1; else x =0;
 if (P2 >= 0)
  y=y+1; else y=100;}
void target ()
{if (x>=10 && y>=10)
   //Target 1 }
```

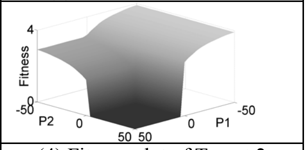
(1) Program fragment

```
//Case study 4 - TeTra
int x,y;
void reset()
  { x = 0; y = 0;}

void target (int P1,P2)
{if (P1 >= 0){
  if (P2 >= 0){
   //Target 2
for(i=1; (x<10 || y<10)
;i++)
  t1(P1,P2); }}}
```

(2) Transformed version



(3) Fitness plot of Target 1



(4) Fitness plot of Target 2

Figure.5 TeTra for finding a feasible path

landscape does not touch the zero surface and so the original scenario: *reset*$\rightarrow$*t1*$\rightarrow$*target* is infeasible. The transformed version shown in Fig. 5-2 tries to construct a feasible path that enables the *target* task to be triggered. Since the input parameters of the affecting *t1* decide whether the assignments are executed, these are included in the transformed version. Once suitable input parameters values are found, a loop of calls is made to the affecting *t1* assignments. The notion of implementing a loop to perform the necessary calls to an affecting function is introduced in [4]. The number of the loop cycles (number of calls) can be determined by reversing the guards of the *target* task. For this case study, this is determined as: loop while (x<10 OR y<10). Similarly, a logical connector AND is reversed to OR and guard operators: $\{<, \leq, >, \geq, =, \neq\}$ are reversed to: $\{\geq, >, \leq, <, \neq, =\}$. Once the affecting functions, the number of calls, and the suitable input parameters values are determined, a feasible path is constructed from the original code by repeatedly calling the affecting functions with the same suitable input parameters values. Fig. 5-4 shows a clear downwards fitness landscape of the transformed version for finding the suitable input parameter values to be applied to *t1*.

Table 1 lists all possible combinations among affected-by and affecting functions. The fields marked by *R* indicate the cases where we conjecture that TeTra is likely to be required. Fields marked by *F/R* indicate that the transformation is only required if the scenario is feasible and fields marked by *N* identify the cases where the transformation is not necessary.

3. Experimental Study and Conclusion

Experiments were performed on the four case studies presented in this paper by using random search

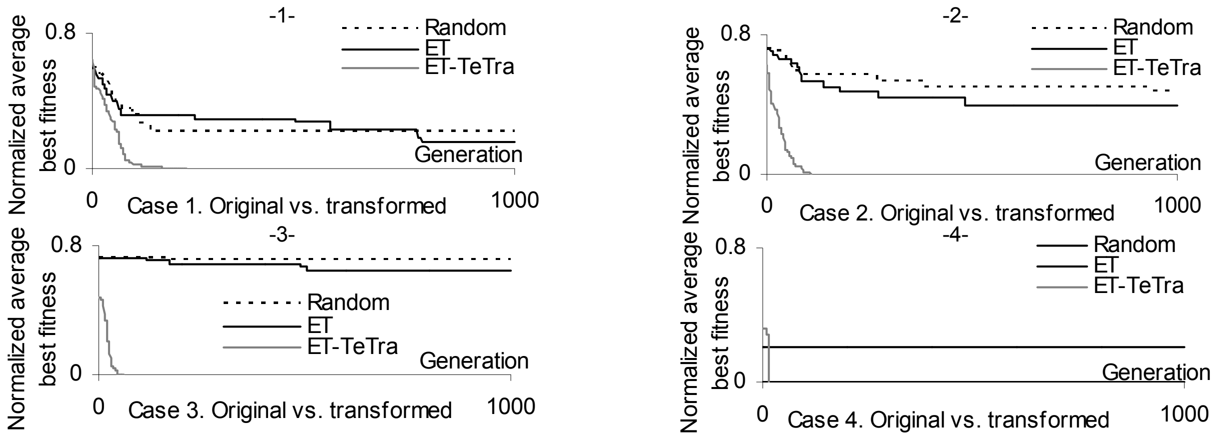


Figure 6: Results of random, ET and ET after TeTra was applied on the four case studies

and the standard ET approach described in Section 1. Although the four case studies are small in terms of code size, the complexity for a search-based algorithm is not related to the code size but it is a function of the search space size [7]. The input domain size used with the four case studies had 4×10^6 possible candidate solutions. Both of the standard ET and random approaches were implemented with the publicly available *GEATbx* [8]. The population size was 50 individuals with two variables in the range [-1000..1000]. The ET methods were: linear-ranking with 1.8 selective pressure, discrete recombination and mutate integer. The search was terminated after 1000 generations or if the objective value of zero was achieved. Finally, each search was repeated 10 times.

Fig. 6 plots the performances of random and ET approaches on each case study before applying the transformation and once again the ET performance after the transformation was applied. Each plot shows the normalized best achieved fitness yet in a specific generation for a particular search approach averaged over ten repetitions of the experiment.

Fig. 6-1, 6-2 and 6-3 plot the performances of the search approaches on Case study 1, 2 and 3 respectively. From these plots, we observe that ET and random searches exhibited relatively similar performance before applying the transformation and they failed to exercise the test target. In contrast, the ET search on the transformed version was successful and hit the target relatively quickly. Fig. 6-4 corresponds to the last case study. Since the untransformed version corresponds to an infeasible path, it was not surprising that ET and random searches both failed. However, the ET performance on the transformed version was very fast in locating the required input values. The empirical results demonstrate that the proposed TeTra approach was effective in improving and enhancing the ET technique. Further research will apply the approach to additional examples and investigate its use to derive feasible paths for the purpose of model-based testing.

Table. 1 Suggesting when TeTra is required

Guard & operator	Assignment (affecting)			
(affected-by)	(op^{pv})	(op^{vv})	(op^{vc})	$(op^{v\pm c})$
g^{pc}, g^{pp}, g^{vp} $(=, <, >, \leq, \geq, \neq)$	N	N	N	N
$g^{vv}(=, <, >, \leq, \geq, \neq)$	R	R	R	R
$g^{vc}(=, <, >, \leq, \geq, \neq)$	R	R	F/R	R

4. References

[1] P. McMinn, "Search-based software test data generation: a survey: Research Articles," *Software Testing, Verification & Reliability*, vol. 14, pp. 105-156, 2004.

[2] J. Wegener, A. Baresel, and H. Sthamer, "Evolutionary test environment for automatic structural testing," *Information and Software Technology*, vol. 43, pp. 841-854, 2001.

[3] N. Tracey, J. Clark, K. Mander, and J. McDermid, "An automated framework for structural test-data generation," presented at Automated Software Engineering, Proceedings. 13th IEEE International Conference on, pp. 285-288, 1998.

[4] M. Harman, "Open Problems in Testability Transformation," presented at Software Testing Verification and Validation Workshop, ICSTW '08. IEEE International Conference on, pp. 196-209, 2008.

[5] M. Harman, L. Hu, R. Hierons, J. Wegener, H. Sthamer, A. Baresel, and M. Roper, "Testability transformation," *Software Engineering, IEEE Transactions on*, vol. 30, pp. 3-16, 2004.

[6] M. Harman and S. Danicic, "Amorphous program slicing," presented at Program Comprehension, IWPC '97. Proceedings, Fifth Iternational Workshop on, 1997.

[7] P. McMinn, M. Harman, D. Binkley, and P. Tonella, "The species per path approach to Search-Based test data generation," in Proceedings of the 2006 international symposium on Software testing and analysis. Portland, Maine, USA: ACM, pp. 13-24, 2006.

[8] H. Pohlheim, "GEATbx - Genetic and Evolutionary Algorithm Toolbox for Matlab," 1994-2008. http://www.geatbx.com.